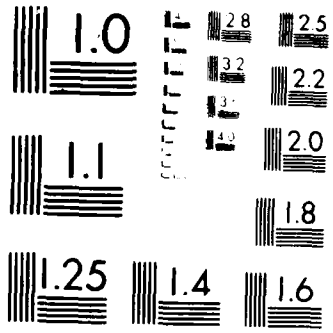
AD-A177 563 AUTOMATED PROBLEM MAPPING: THE CRYSTAL RUNTIME SYSTEM 1/1
U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE
J H SALTZ ET AL. JAN 87 YALEU/DCS/RR-510
UNCLASSIFIED N00014-86-K-0926 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART

Automated Problem Mapping: the Crystal Runtime System

Joel H. Saltz, Marina C. Chen

Research Report YALEU/DCS/RR-510
January 1987

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract: Very high level language algorithm specification promises to be a crucial factor in the enhancement of softwate reliability. The ability of this system to map problems onto very large message passing machines in an automated manner should greatly increae the utility of high performance architectures. These architectures may be quite cost effective from the hardware point of view, but unattactive due to the difficulty of providing software able to exploit the potential of the machines.

Automated Problem Mapping: the Crystal Runtime System

Joel H. Saltz, Marina C. Chen

Research Report YALEU/DCS/RR-510
January 1987

Work supported in part by NSF grant no. DCR 8106181 and the Office of Naval Research under contract no. 00014-86-K-0926.

1 Overview

The effective utilisation of multiprocessors, particularly those with architectures that cannot support shared memory in an efficient way, is currently dependent on the ability of the user to map the problem onto the multiprocessor. In order to obtain high levels of efficiency, this mapping must distribute computational load relatively evenly between the machine's processors and must *minimise the effects of interprocessor* communication delay on algorithm performance. The need to explicitly designate a problem decomposition and to verify that the decomposition is both correct and has the desired performance characteristics can be a time consuming and error prone task. In cases in which the load distribution of an algorithm cannot be predicted sufficiently well in advance to allow a deterministic decomposition to be specified, it may be necessary to specify a family of problem decompositions along with a procedure for run time load management. While the development of methods for dynamically balancing loads is an active area of current research (e.g. see [27] [14] [5] [10] [22] [30]), without the development of automated mapping methods, the implementation of such schemes can be particularly time consuming and difficult [26].

A methodology will be developed that can insulate the user from the considerations required to produce efficient programs for multiprocessor machines while still enabling the user to achieve high levels of performance. Programs will be written in a very high level programming language Crystal.

1.1 Language and Compiler

Existing approaches for programming parallel machines can be broadly categorised into two groups: (1) devising parallelizing compilers for imperative languages (e.g. Fortran) such as the Bulldog compiler [12,11] and others [21], [2], and (2) devising parallel language constructs and expressing parallelism explicitly by programmers. The first approach has the advantage that programs can be written in familiar languages and existing programs can be transformed by parallelising compilers for execution on the new machines. One difficulty that has been encountered in the attempt to exploit large scale data level parallelism in conventional languages is the need to perform extensive analysis due to interprocedural dependencies. However, useful parallelism may be lost because the programmer may specify unneeded sequentialisation when writing programs in a imperative language.

Another difficulty is that the parallelism discovered this way is limited by the algorithm embodied by the program. It is unlikely that the transformations provided by the parallelising compiler are sophisticated enough for the task of redesigning programs better suited for parallel processing. Consider, for example, the problem of sorting. Quicksort is a very good sequential method which can be parallelised by spawning a process for each of the two recursive calls that this sorting algorithm must make. The time complexity is indeed improved from $O(n \log n)$ (average case) in the sequential version to $O(n)$ (since $O(n)$ comparisons are needed at the top level and the number of comparisons is halved at every level thereafter) by using $O(n)$ independent parallel processes. However, what can be achieved by various parallel sorting networks (e.g., [32]) with $O(n)$ processors is $O(\log^2 n)$, which is significantly faster for large n. Numerous other good sequential algorithms have the same property that they do not lend themselves to efficient parallel implementations, as exemplified by many of the newly devised parallel algorithms [9] which are significantly different from their sequential counterparts.

This point leads to the second approach — parallel programming, where parallelism is explicitly expressed in a program. This is flexible enough to be applied to either class of parallel machines (shared-memory machines or message-passing machines) as well as any kind of parallel algorithm. However, parallel programming and debugging can be extremely difficult with thousands of interacting processes. Most parallel languages, either proposed or in use, have explicit constructs for parallelism. Programmers specify how tasks should be partitioned and which ones can be run in parallel (e.g., *futures in Multilisp* [17,18], "in" and "out" in Linda [1]), or they specify explicitly the communication between processes (e.g., "?" and "!" in CSP [19]). The specification of communication is very tricky because it requires the programmer to keep track of both a processor's own state and its interactions with other processes. Explicit task decomposition by the user may yield inefficient code for a large class of problems for which an efficient decomposition cannot be

known until run-time.

A Crystal program is a very high level algorithm specification in which the detailed interactions among processes in space and time are suppressed. The Crystal compiler and runtime system allow the generation of instructions to direct an assemblage of communicating processors in the efficient execution of the specified algorithm.

Crystal is a very-high-level language in which a user program resembles a concise, formal mathematical description. The language Crystal appears to be quite straightforward to use yet is designed to have the modularity and freedom from side effects that has been shown to be of substantial benefit in the automatic detection of parallelism. For an overview of the language Crystal, language constructs, and programming examples, please see [6]. No explicit passing of messages is needed in the program specification, and task decomposition is done automatically by the Crystal compiler. The compiler generates as many logical processes as possible and then combines clusters of logical processes to produce a problem decomposition that possesses a degree of granularity that is appropriate for the target machine. In cases in which the pattern of computations in a given section of the program is known at compile time, a direct mapping of the algorithm in question may be performed. When the pattern of computations are fully determined only at runtime, the compiler constructs a symbolic representation of the data dependencies. This symbolic representation is used by a run-time system which aggregates the required computations. When enough regularity is present the runtime system creates a parameterised mapping scheme. Different instances of the mapping scheme has a range of properties. Using knowledge of target machine characteristics, the runtime system chooses the appropriate instance of the mapping scheme and dynamically maps the computations onto the target architecture.

1.2 Load Balancing Analysis

The run time system must be designed to require low computational overhead. The way in which the runtime system functions is greatly dependent on the amount of regularity that is detected in a given portion of a crystal program. In many scientific problems, the computations to be performed by a given procedure may be determined in advance once the main data structures of the problem are set up. This set up phase will very often occur at run time. An example of such a code is the preconditioned congugate gradient type linear equation solver described below.

A variety of strategies are used to contain the cost of the run time system in these relatively predictable problems. As much information as possible is obtained from the analysis performed during compilation and when possible, a parameterised mapping function is produced from this information. This mapping function describes how the computation could be mapped onto the processors of a machine and how those computations within each processor would be scheduled. The space of possible problem partitionings is consequently greatly constrained; the mapping chosen is the one that is estimated to give the best results for the given problem and the target machine.

When a given computation must be repeatedly performed, as it might be in an iterative algorithm or in the solution of a time dependent partial differential equation, a problem may be remapped from time to time in search of better performance. Finding the best parameters for mapping given a repetitively executed procedure will often be an iterative process. Performance evaluation mechanisms will be provided to weigh the costs of remapping against possible performance improvements that could be obtained [27],[24], [25]. The control of the remapping can take advantage of the fact that the algorithm is iterative, we can hence employ statistically formulated run time policies for the control of remapping that attempt to minimise expected costs under conditions of uncertainty [27].

There are a large number of problems in which the patterns of data dependency or the distribution of work change during the evolution of the problem. Many of these problems contain enough regularity to make the use of a parameterised problem mapping procedure likely to be advantageous. Examples of such problems include adaptive mesh refinement, particle methods, intermediate and high level image processing and time driven discrete event simulations [4] [3] [24] [16] [15] [28]. The use of policies for dynamic load balancing are vital for the efficient multiprocessor solutions of these problems.

To illustrate the issues involved in balancing load in gradually changing systems, consider the implementation of a time driven discrete event simulator on a message passing multiprocessor. At varying points in the simulation different areas of the simulated domain will have varying levels of activity. These domain regions will consequently require differing amounts of computational work. During the course of the simulation, the majority of the interactions between portions of the domain will be local in character. Consider the tradeoffs that are made in choosing a mapping of domain to processors. A domain decomposition that assigns contiguous regions to processors will lead to relatively low communication costs but may require frequent remappings in order to preserve a good balance of load. Domain decompositions that assign many smaller non contiguous regions of domain to each processor will be subject to higher communication costs but will be less subject to performance deterioration due to developing load imbalances.

A systematic set of policies must take into account tradeoffs between costs of communication and the propensity for loads to become unbalanced, and must also take into account the tradeoffs between the costs and benefits obtained by balancing load at a given point in the computations [24].

There exist classes of problems whose patterns of computations are so irregular that any sort of a-priori problem mapping is likely to be ineffective. Algorithms which attempt to exploit parallelism at the expression level rather than through data decomposition appear to be particularly problematic in this respect. There is a considerable research effort directed towards the investigation of load balancing strategies that could prove to be effective in these less structured contexts [8] [29] [31] [13] [10] [23]. A distributed branch-and-bound mechanism extending some of these principles has been devised as part of the Crystal run-time system for supporting expression level parallelism. Expression parallelism is obtained by interpreting function calls as processes: function calls that do not have any dependency can be executed in parallel. Generally speaking, problems that have fast parallel algorithms (in NC) exhibits a large scale of data level parallelism while NP hard problems exhibits a high-degree of expression parallelism and very little data level parallelism. There exists a potential difficulty in utilising expression level parallelism: the total number of computations that must be performed may increase as one attempts to gain more parallelism. Many of these computations involve searches of one sort or another; parallelism may be obtained though the investigation of alternatives that the sequential version would be able to identify as being unpromising. This can lead to the anomaly that the sequential processing may out-perform the parallel one because useless computations are performed in the parallel version. It is for this reason branch-and-bound becomes important in a general-purpose programming environment that supports parallelism automatically. The scheduling of work in this case cannot in general be carried out in advance even in a rough sense, and diffusion based load balancing is consequently used.

1.3 Runtime System Control

In the design of a system to automate the run time mapping of work to processors, a number of interacting factors that determine performance must be taken into account. Virtually all non trivial programs are modular in nature. In these programs the computations may be conceived of as occurring in a number of phases. For instance, during the course of computations, a scientific code may calculate a Jacobian, perform local relaxations, perform FFTs, or calculate inner products. We have made the tacit assumption so far that problems consist only of one phase. When problems consist of a sequence of phases, the mappings that are suitable for a given phase cannot be expected to be suitable for all phases. At these interphase transitions, there will often be a choice available between 1) remapping the problem so that suitable mappings may be used for each phase, or 2) avoiding at least some of the work associated with remapping by using mappings that for some phases that may be less efficient [7]. When data dependencies allow the coscheduling of two or more phases, the runtime system must decide whether using this expression level parallelism will be advantageous. Determining the best course of acti n requires taking into account the costs of remappings, the costs of executing various phases given the mappings under consideration, and the pattern of phase executions. The costs of executing phases given various mappings will be dependent on the run time parameters defining the patterns of computation and the size of the problem to be solved. The pattern of phases may also not be fully pre-determined.

2 Programming in Crystal

Crystal is a general pupose language, and it is particularly powerful for parallel computing in the sense that all fast parallel algorithms — characterised by the class NC [9] (polynomial number of processors and polylogarithmic time complexity) — can be specified in Crystal. A description of Crystal, a discussion of the compiler along with examples of Crystal syntax may be found elsewhere in these proceedings [6].

3 Analysis of a Model Problem

We present a sparse matrix forward solver as a model algorithm. This example is used to illustrate some of the considerations involved in the design of automated problem mapping methods. This solver is taken from a crystal program implementing a preconditioned conjugate gradient type method, orthomin(1) with incomplete LU preconditioning, used to solve sparse linear systems arising from discretisations of partial differential equations. The variables in this problem represent function values at grid points in a two dimensional partial differential equation. The stencil used to discretise the partial differential equation establishes data dependencies between the unknowns.

The incomplete factorisation of the matrix A produces sparse lower and upper triangular matrices L and U. One of the crucial features in the parallel implementation of conjugate gradient like methods using this incomplete LU preconditioner is the efficient implementation of the forward and backward solutions involved in the preconditioning. We will restrict attention to the solution of the lower triangular system, considerations involved in the solution of the upper triangular system are virtually identical.

A Crystal program that implements the forward solve is given below:

```
!  forward solve:  solving L x = y where L is a sparse lower triangular matrix
!  L is in the lower triangle of B

fsolve(B, y) = [x(i) | i = 1:n]
where (
   x(i) = y[i] - (\+ { B(i).value(k) * x(B(i).col(k))
      (k = 1:B(i).ncol) and ( B(i).col(k)<i) })
      )
```

A sparse matrix is represented by a tuple A of records, where each record A(i) represents a row of the sparse matrix and contains three fields. The three fields [A(i).ncol, A(i).col, A(i).value] contain respectively the number of non-sero elements in that row, the column numbers for these elements, and the values of the elements. The notation of a generated set [6] is used in the definition of fsolve and is represented by pairs of curly braces. The notation of \+ in front of a set of elements means the sum over those elements. Any binary associative operator $\oplus$ applied to a set of elements can be expressed by $\backslash\oplus$. Using the set notation instead of a do loop has the advantage that sequentiality is not introduced due to the semantics of the programming construct. The absence of such sequentiality allows the straightforward evaluation of the operation to take advantage of the maximum parallelism possible.

3.1 Symbolic DAG Generator

In many scientific problems including the sparse substitution algorithm described above, the computations to be performed by a given procedure may be determined in advance once the main data structures of the problem are set up. A symbolic representation of a directed acyclic graph (DAG) representing the data dependency of an algorithm can be produced by the compiler. For instance, in the subprogram defining the forward solve fsolve the symbolic DAG generator produced by

the the compiler looks like the following where nodes is the set of vertices of the DAG and edges is the set of directed edges [u,v] pointed from from vertex v to vertex u:

```
nodes = {i | i = 1:n}
edges = \union {[ i,B(i).col(k) ] | (k = 1:B(i).ncol) and (B(i).col(k) < i), i = 1:n}
```

When matrix B become available either at compile time or at run time, the explicit DAG can be produced.

3.2 Load balancing: from algorithm designer's view

In the simplest form of incomplete LU preconditioning, the factors L and U have the same sparsity structure as the lower and upper portions of A respectively. A prior knowledge of the sparsity structure will be used to advantage in the generation of the following parameterised problem mapping. Note that this prior knowledge will not be needed when the automated version of the problem mapping is used. This automated version of problem mapping will be described in the next section.

We will assume that we have a rectangular array of grid points, all points are connected with the same stencil. The stencil is assumed to link a given point with it's left, right, upper and lower neighbors in the grid. The matrix is formed by using the so called natural ordering in which grid points are numbered in a row-wise fashion beginning with the first column of the first row of the domain. We assume the same stencil is utilised for all mesh points in the problem. Again, note that these assumptions are used in the description of the explicitly defined parameterised mappings of grid points to processors. The automated mapping techniques to be discussed later is able to directly utilize an arbitrary lower triangular matrix. The automated methods produce the same mappings as the grid oriented techniques to be discussed below when the lower triangular matrix is a representation of a grid that is amenable to the grid oriented techniques.

The data dependency pattern between unknowns in the lower triangular solution may be best understood by refering back to the stencil and the grid utilised in the formulation of the problem [33]. Let $x_{i,j}$ be the location of a mesh point in the two dimensional domain, where $1 \leq i \leq n$ and $1 \leq j \leq n$. In the definition of the problem, a function value at a point $x_{i,j}$ is linearly dependent on function values at a given set of surrounding points. When a system involving a lower triangular matrix with the same sparsity structure as A is solved, the only interactions that need be considered are with variables in the grid that are in rows before i, as well as variables in row i that are before column j.

The grid points in a given row must be solved for sequentially, due to the coupling of each point to it's immediate neighbors. We assume that the stencil is rather small, so that relatively few calculations are involved in obtaining the value for a single grid point of the domain. In these mappings, the smallest unit of work that may be assigned to a particular processor consists of the computations pertaining to a particular row of grid points. The computations in a given row i depend only on results from row $j < i$. Depending on the relative size and properties of the problem and of the machine, better performance may be obtained by using a coarser grained assignment of work in which contiguous blocks of several rows are assigned to each of k processors. When there are more blocks of rows than there are processors, a wrapped assignment is used in which blocks are assigned to processors modulo k.

Given a fixed assignment of grid points to processors, one may be free to schedule the work associated with calculating values at mesh points in a variety of different ways. This processor scheduling has a marked effect on the frequency with which processors must interact to exchange information. When a five point stencil is utilised, a convenient method of scheduling is to partition each block into windows of w columns each. Because of the use of the five point stencil, values for all points in a given window of a block may be computed before any work on the next window is begun. If one numbers the windows in each block from left to right, block i may commence work on window j when block $i-1$ has finished work on window j. This leads to a pattern of computation [33] in which a wavefront of computation is seen to propagate from the lower left portion of the domain (Figure 1). The block size and the size chosen for the window both determine the coarseness of the computation's granularity. A quantitative analysis of this tradeoff in the case where the

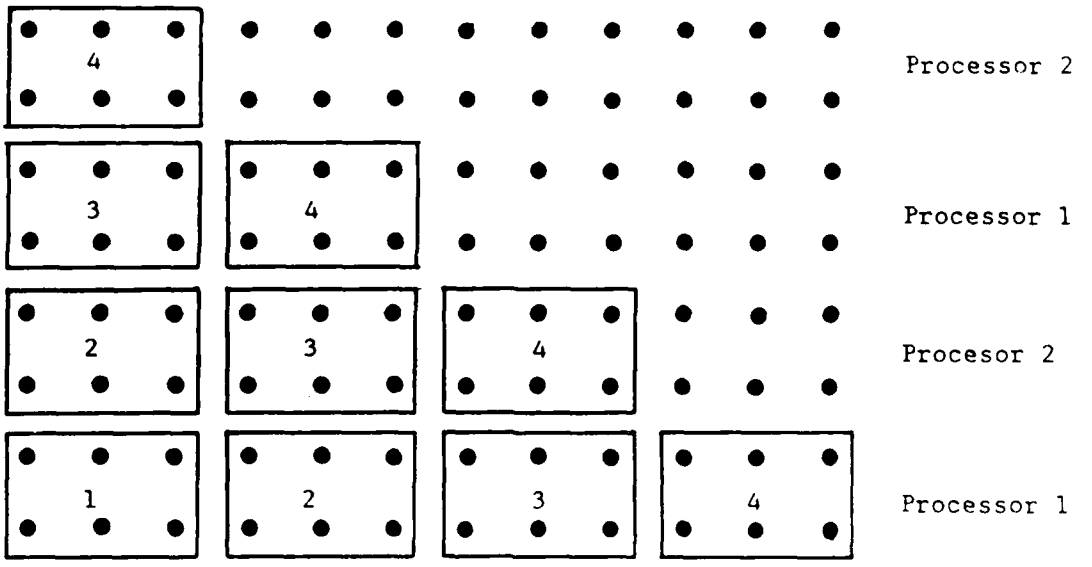


Figure 1: Two processors, five point stencil, block size = 2, window = 3. Numbers designate computational phases.

block size is equal to the window size is given in [33]. For a grid whose points are connected by an arbitrary stencil, the definition of work schedules that maintain data dependency relations yet allow for varying degrees of granularity is somewhat more subtle. Work is begun in the first row of the first block, and in this row the values for w window of grid points are calculated. Following this, values are found for all mesh points in the block for which data dependencies allow calculation. The computation proceeds after this in stages, with the computations that may proceed in a block at a given time being determined by dataflow considerations. Now for an arbitrary but uniform stencil, the computation of the variable at row i column j may require data from rows $i-q$, columns $j-v_q$, for $1 \leq q \leq i-1$, $v_q \geq 1$. Thus if one wishes to aggregate points in blocks into larger units, with each unit to be calculated sequentially, the partitioning will take on a sig-sag form. Figure 2 depicts the pattern of wavefronts that results from partitioning a domain with a nine point stencil into blocks of size two, and scheduling computation using a window size of two.

3.3 Automated load balancing

In order to automate problem partitioning and work scheduling, it is essential to be able to dispense with as much application dependent information as possible. We have developed and tested a method for generating a parameterised mapping function to partition work possessing data dependencies given by a directed acyclic graph (DAG) generated by the DAG generator of Section 3.1 . In the following, the methods for parameterised mapping will be discussed in the context of solving the lower triangular system of linear equations we have been considering. The methods to be described utilise only the sparsity structure of the lower triangular matrix in the generation of the mapping and scheduling function, no representation of a physical domain is required. Algorithms for inexpensively generating parameterised mappings of acyclic graphs that allow for good performance will be crucial to the overall effectiveness of the crystal run time system.

We will assume that all computations pertaining to a row of the matrix will be assigned to a single processor. Note that this implies a potentially fine degree of granularity as we are dealing with matrices having few non zero elements in a row. The concurrency achievable through the use of this algorithm is largely determined by the dependencies between the rows of the lower triangular matrix. The approach to be taken here is to utilise an analysis of the data dependencies

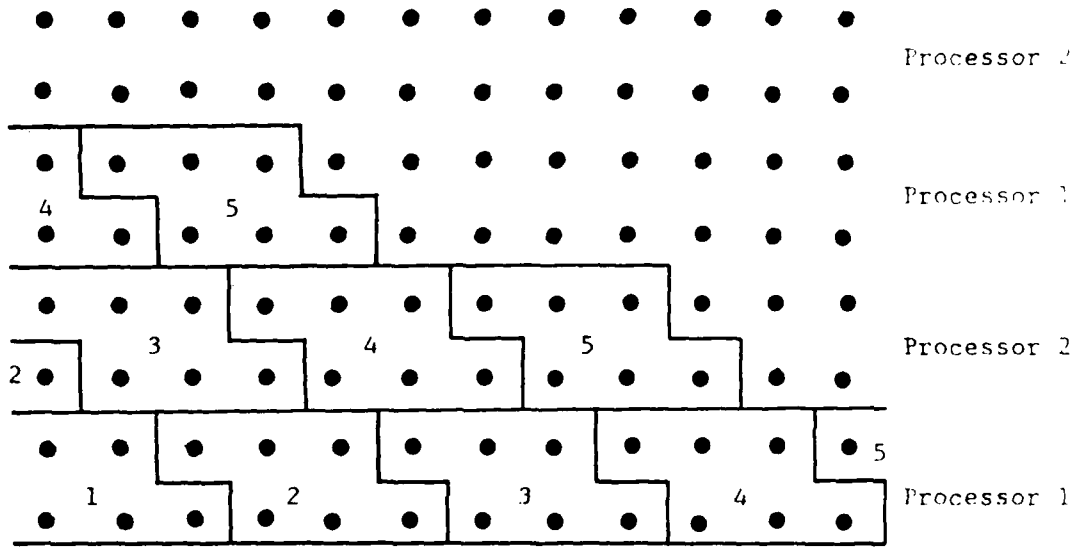


Figure 2: Two processors, nine point stencil, block size = 2, window = 3. Numbers designate computational phases.

to produce a parameterized mapping. This partitioning process will take place in two stages. The rows of the L will be partitioned into a number of disjoint sets to be called strings. The strings will then be distributed between processors with all computations pertaining to a given string assigned to a single processor.

The partitioning of the rows into strings and the distribution of the strings are performed so as to attempt to satisfy the objectives of maximizing potential concurrency and of minimizing communication costs. In order to reduce the amount of interprocessor communication, we want strings to consist of rows with data dependencies. So as not to compromise potential concurrency, *strings should contain only rows* that could not under any circumstances be evaluated concurrently.

3.4 Wavefront and String Generator

A general description of how the rows may be partitioned into strings will now be given, details of the algorithm are presented elsewhere [26]. The order in which variables, described by rows in L, can be solved may be depicted by a directed acyclic graph D. The evaluation of rows in the L are represented by the vertices of D, and the data dependencies between the rows by the D's edges. The dependence of matrix row a on matrix row b is represented by an edge going from vertex b to vertex a. A topological sort may be performed which partitions to DAG into wavefronts. This sort is performed by alternately removing all vertices that are not pointed to by edges, and then removing all edges that emanated from the removed vertices. All vertices removed during a given stage constitute a wavefront; the wavefronts are numbered by consecutive integers. An adaptation of a common topological sort algorithm [20] allows the wavefronts of a DAG to be calculated efficiently.

Strings are chosen to be in a rough sense orthogonal to the wavefronts of the DAG D, and so that the graph describing the *inter-string* data dependencies is a directed acyclic graph, to be called here the string DAG. Choosing strings in this way greatly increases the flexibility allowed in the scheduling of computation It should be noted that in cases where the lower triangular matrix in question has been obtained from a rectangular array of grid points with a stencil in the manner described above, the partitioning process will assign a row, a column or a diagonal of grid points to each string. The directed acyclic graph describing inter-string data dependencies is in this case simply a chain with only nearest neighbor relationships. Figure 3 shows the wavefronts and strings assigned to the matrix representing a 12 by 8 point domain with an eight point stencil.

The string DAG may be distributed among processors in a variety of ways. Mapping large

POINTS

85	86	87	88	89	90	91	92	93	94	95	96
73	74	75	76	77	78	79	80	81	82	83	84
61	62	63	64	65	66	67	68	69	70	71	72
49	50	51	52	53	54	55	56	57	58	59	60
37	38	39	40	41	42	43	44	45	46	47	48
25	26	27	28	29	30	31	32	33	34	35	36
13	14	15	16	17	18	19	20	21	22	23	24
1	2	3	4	5	6	7	8	9	10	11	12

WAVEFRONTS

15	16	17	18	19	20	21	22	23	24	25	26
13	14	15	16	17	18	19	20	21	22	23	24
11	12	13	14	15	16	17	18	19	20	21	22
9	10	11	12	13	14	15	16	17	18	19	20
7	8	9	10	11	12	13	14	15	16	17	18
5	6	7	8	9	10	11	12	13	14	15	16
3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12

STRINGS

8	8	8	8	8	8	8	8	8	8	8	8
7	7	7	7	7	7	7	7	7	7	7	7
6	6	6	6	6	6	6	6	6	6	6	6
5	5	5	5	5	5	5	5	5	5	5	5
4	4	4	4	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	1

Figure 3: Points, wavefronts, and strings for a 12 by 8 point domain, eight point stencil

contiguous sections of the string DAG onto each processor will tend to minimise communication costs but will also tend to lead to poor load distributions. Scattering or wrapping strings that are contiguous in the DAG may lead to a much better load distribution at the price of increased communication costs [26].

The work associated with each cluster of strings may be scheduled with varying degrees of granularity. The string DAG defines a partial ordering among the strings. The starting strings may be defined as the strings that precede all others in this partial ordering. Computations of rows in these strings are not dependent on information from any other strings in the string DAG.

The granularity of parallelism may be determined by fixing the amount of work starting strings can perform before communicating their data to other strings in the string DAG. Simple relationships involving the wavefronts of *rows* allows the calculation of which rows may be solved for by a processor assigned to a cluster of strings.

The details of the rules for scheduling rows depend on the methods for aggregating strings used by the parameterised mapping function. One particularly straightforward method is to perform a topological sort on the string DAG in order to impose a strict ordering on the strings. Once this ordering is performed, contiguous blocks of strings having constant size b are demarcated, and are assigned to consecutive processors in a wrapped manner.

We make the following observation, with a simple proof by induction given in [26]:

Proposition 1 *Assume we partition the computation into phases p and allow the first string in the first block to compute values for rows between wavefronts* $(p-1)w+1$ *and* pw. *At the end of phase p, all strings in block i may calculate values for rows having wavefronts* $(p-i)w+1$ *to* $(p-i+1)w$.

It is shown [26] that in the special case in which the linearly ordered set of strings have only nearest neighbor data dependencies, the largest wavefront that can be computed by the jth string in block i, during phase p is $w(p+1-i)+(i-1)b+j-1$.

As is documented with experimental work in [26], both the way in which strings are assigned to processors and the degree of granularity in the scheduling of computations within strings influence both load balance and communication costs. Once the decomposition into wavefronts and strings is performed, one has available a range of mapping and scheduling schemes which can be utilized in a way that is appropriate for a given machine and problem.

4 Summary

Very high level language algorithm specification promises to be a crucial factor in the enhancement of software reliability. As described above, we propose an integrated system which has the promise of greatly facilitating the development of large complex high performance software systems. These programs would be executable in an efficient manner on massively parallel architectures. Fault tolerance is an inherent part of the system design since the system incorporates dynamic runtime problem mapping. This mapping allows for execution time system reconfiguration.

References

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelerter. Linda and friends. *IEEE Computer*, August 1986.

[2] F Allen. Compiling for parallelism. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, page 126, October 1985.

[3] M. J. Berger and A. Jameson. Automatic adaptive grid refinement for the euler equations. *AAIA Journal*, 23:561–568, 1985.

[4] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–512, 1984.

[5] S. Bokhari. *Partitioning Problems in Parallel, Pipelined, and Distributed Computing.* Report 85-54, ICASE, November 1985.

[6] M. C. Chen. Very-high-level parallel programming in crystal. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[7] T. F. Chen. *On Gray Code Mappings for Mesh-FFTs on Binary N-Cubes.* Report 86.17, RIACS, September 1986.

[8] W. W. Chu, L. J. Holloway, and K. Efe M. Lam. Task allocation in distributed data processing. *Computer*, 13(11):57–69, November 1980.

[9] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, (64):2–22, 1985.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Eng.*, SE-12:662–675, May 1986.

[11] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures. The ACM Doctoral Dissertation Award Series*, The MIT Press, 1985.

[12] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, Association for Computing Machinery, June 1984.

[13] G. J. Foschini. *On Heavy Traffic Diffusion Analysis and Dynamic Routing in Switched Networks.* North-Holland, New York, 1977.

[14] W. Gropp. *Dynamic Grid Manipulation for PDEs on Hypercube Parallel Processors.* Department of Computer Science YALEU/DCS/TR-458, Yale University, March 1986.

[15] W. Gropp. *Dynamic Grid Manipulation for PDEs on Hypercube Parallel Processors.* Department of Computer Science YALEU/DCS/TR-458, Yale University, March 1986.

[16] W. Gropp. *Local Uniform Mesh Refinement on Loosely-coupled Parallel Processors.* Department of Computer Science YALEU/DCS/TR-352, Yale University, December 1984.

[17] Jr. Halstead, Robert H. Multilisp: a language for concurrent symbolic computation. *ACM Transaction on Programming Language and Systems*, October 1985.

[18] Jr. Halstead, Robert H. Parallel symbolic computing. *IEEE Computer*, August 1986.

[19] C.A.R. Hoare. Communicating sequential processes. *Communication of ACM*, 21(8):666–677, 1978.

[20] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, Rockville Maryland, 1983.

[21] K. Kennedy. Compilation for n-processor architectures. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, page 15, October 1985.

[22] S. H. Bokhari M. A. Iqbal, J. H. Saltz. Performance tradeoffs in static and dynamic load balancing strategies. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[23] L. M. Ni, C. Xu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Trans. on Software Eng.*, SE-11:1153–1161, October 1985.

[24] D. Nicol and J. Saltz. *Dynamic Remapping of Parallel Computations with Varying Resource Demands*. Report 86-45, ICASE, July 1986. submitted to Transactions on Computers.

[25] D. M. Nicol and Jr. P. F. Reynolds. *An Optimal Repartitioning Decision Policy*. Report 86-7, ICASE, February 1986.

[26] J. Saltz. Methods for automated problem mapping. In *Proceedings of IMA Workshop on Numerical Algorithms for Parallel Computer Architectures*, 1987.

[27] J. H. Saltz and D. M. Nicol. *Statistical Methodologies for the Control of Dynamic Remapping*. Report 86-46, ICASE, July 1986. to appear in the Proceedings of the Army Research Workshop on Parallel Processing and Medium Scale Multiprocessors, Palo Alto, California, January 1986.

[28] R. Smith and J. Saltz. Performance analysis of strategies for moving mesh control. In *Proceedings of the CMG XV International Conference on the Management and Performace Evaluation of Computer Systems*, pages 301–308, 1984.

[29] J. A. Stankovic. An application of bayesian decision theory to decentralized control of job scheduling. *IEEE Trans. on Computers*, C-34(2):117–130, February 1985.

[30] A. N. Tantawi and D. Towsley. Optimal static load balancing. *Journal of the ACM*, 32(2):445–465, April 1985.

[31] D. Towsley. Queueing network models with state-department routing. *Journal of the ACM*, 27(2):323–327, April 1980.

[32] Jeffrey D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[33] M. Schultz Y. Saad. *Parallel Implementations of Preconditioned Conjugate Gradient Methods*. Department of Computer Science YALEU/DCS/TR-425, Yale University, October 1985.

END

4-88

DTIC